Computer Science 134C
Introduction to Computer Science, in Python
Lecture #26 (Decoration)
November 14, 2018

Keywords
annotation, count, decoration, memoize

An introduction to decoration.

1. Questions?
2. We now look into the concept of *decoration* (other languages call this *annotation*).
   (a) In Python, function (and class) definitions can be *decorated*. The process of decoration allows one to—just before a function definition is bound to its name—make modifications to the definition.
   (b) These modifications are accomplished by functions that are called *on a function, when it is defined*. These function applications are signified by the use of an at-sign (`@`).
   (c) Decorators take a single parameter—the function or class they are decorating—and return an alternative function that is to be used as the definition.
   (d) For example, suppose we define `tag`, a function that adds a new attribute to a function:

```
def tag(f):
    f.secret = 42
    return f
```

   We can then perform the following:

```
def sqr(n):
    return n*n
sqr = tag(sqr)
```

   At this point, the `sqr` function now has a `secret` attribute, 42.
   (e) We can perform this tagging operation transparently, when the function is defined:

```
@tag              # modifies the definition of sqr
def sqr(n):
    return n*n

print(sqr.secret) # prints 42
```

   (f) Example: a decorator that adds a counter that keeps track of function calls.

```
def count(f):
    def wrapper(*args):
        wrapper.counter += 1
        return f(*args)
    wrapper.counter = 0
    return wrapper
```

With `count`, we can keep track of the number of times a function is called.

(g) Example: Memoization. We construct an `@memoize` decorator that adds a dictionary to a function to manage memoization.

(h) Decorators can be composed!

3. Application: The edit distance between strings.

(a) It's often useful to know how "far apart" two strings are:

i. How much effort would be required to edit one string into another?

ii. How similar are two strands of DNA?

iii. Is there significant similarity between two pieces of Python code?

(b) Ahead of time, we keep track of the types of *edit operations* we might allow and associate with each, a cost. For example, in converting string `a` into string `b` we might allow *insertion of letters of* `b` of the *deletion of letters from* `a`. (It's easy to see that this is sufficient: delete all the letters of `a`, then insert the letters of `b`.) Perhaps each "costs" one unit of effort.

(c) We should allow the preservation of letters of `a` that are shared with letters of `b`. We might think of this a free *copying* operation.

(d) We can then write a recursive method, `dist`, that computes the minimum cost of editing `a` into `b`.

(e) There are *lots* of different possible edits to check, but with a bit of *memoization* we can leverage common sub-problems.

⋆